by
Ray Duncan

Power Programming

Dynamic Data Exchange and the Clipboard in Windows

■ **As an interprocess communications facility, DDE is better than nothing, but it suffers from severe limitations at both the user level and the programmer level.**

Although *Microsoft Windows* 3.0 has two interprocess communications (IPC) mechanisms—the clipboard and Dynamic Data Exchange (DDE)—both of these mechanisms are presented by applications at such a high level of abstraction that the average user has no conception whatsoever of where they're implemented or how they work. This is as it should be. Unfortunately, as a software developer, you can't safely coast along in the same blissful ignorance as the average *Windows* user. You need to understand how the primitive DDE components we've discussed in the last two installments map onto the "preferred" user interface for *Windows* applications—and the mapping is nontrivial.

Let's consider some typical *Windows* IPC scenarios. First, suppose I have some data in an *Excel* spreadsheet and I want to incorporate a graph based on that data into a *Microsoft Word for Windows* document. I launch *Excel* (or activate *Excel* by clicking on its window or on its name in the Task List), load the *Excel* spreadsheet, select the data, and create a new chart via *Excel*'s File New Chart menu sequence. After tuning up the chart with *Excel*'s fancy formatting functions, I select the whole chart by clicking outside the plotting area and copy it to the clipboard with the Edit Copy menu sequence. Finally, I load or activate *Winword,* open my document, position the insertion point for the chart by clicking the mouse pointer at the appropriate place within the text, and paste the graphic into the document using *Winword*'s Edit Paste menu sequence or with the key combination Shift-Ins (see Figure 1).

So far, everything seems pretty simple. By observing how applications behave, I get a vague mental image of the *Windows* clipboard as a sort of pigeonhole for data that is maintained by the system. When I issue Copy or Cut commands, I can easily imagine that the application reacts by making a function call to *Windows,* passing the address of the selected data, and that *Windows* merely puts a copy of the data into the pigeonhole and then returns from the function call. Similarly, when I use a Paste command, I might deduce that the application makes a function call to *Windows* and gets back an address for the pigeonhole, after which it can proceed to copy data from the pigeonhole to its own workspace. It almost seems like a joke to refer to the clipboard as an IPC mechanism at all, because it's totally passive—nothing gets communicated without active intervention from the user at both ends of the transaction.

The straightforward copy-and-paste sequence I just used is very convenient until I need to correct or extend the original data that the graph was based on. If I put a new value into the *Excel* spreadsheet, the *Excel* graph is updated immediately, but the graph in the *Winword* document doesn't change. This certainly seems reasonable: There's no obvious way *Winword* could know where the graph on the clipboard came from, or that the original version had been altered. But having faith that there ought to be a better way to get this job done, I browse through the *Winword* manual and come across a reference to data *links,* which sound like

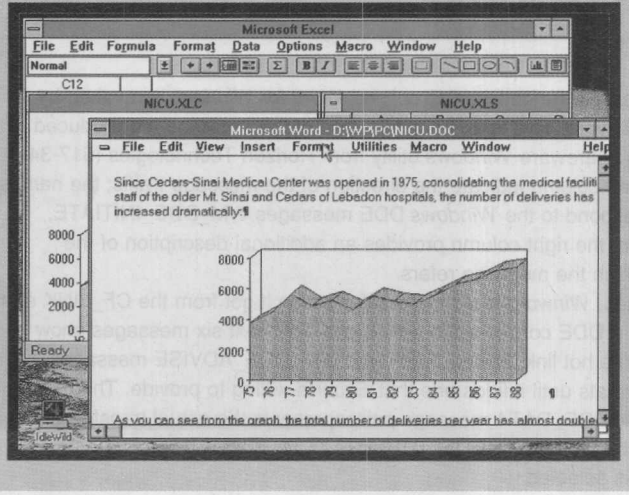

Figure 1: An example of a hot link between *Excel* and *Word for Windows.* Data is entered into an *Excel* spreadsheet, a graph is created, the graph is copied to the clipboard, and then the *Winword* Edit Paste Link command inserts the graph into a word processing document.

they could eliminate a lot of repetitive copying and pasting. I return to *Excel*, select the graph and use the Edit Copy sequence as before, switch to *Winword*, load my document, set the insertion point, and then invoke Edit Paste Link rather than Edit Paste.

The result is gratifying: the graph magically appears in the document, just as it did in our first experiment, but now, whenever I activate *Excel* and change the spreadsheet data, alterations in the graph are immediately reflected in the *Winword* document, too. The unavoidable conclusion is that a DDE transaction is going on between *Excel* and *Winword*, and a little spelunking with the shareware utility DDEWatch shows, sure enough, that the Paste Link operation triggers a whole flurry of DDE messages (see Figure 2).

Our previous mental model of the *Windows* clipboard as a dumb pigeonhole for some data doesn't account for this application behavior at all. Even though we used the exact same Edit Copy command at the *Excel* end of things on both occasions, the *Winword* Paste operation simply sucked in a static image, while the *Winword* Paste Link operation was somehow handed enough information to set up a DDE conversation with *Excel* as the server. Evidently, the *Windows* Clipboard (like nearly everything else in *Windows*) is a lot more complicated than it appears at first glance.

THE CLIPBOARD AND DDE

The first thing to realize is that the *Windows* Clipboard utility program, which appears in the Program Manager's Accessories program group, bears almost no resemblance to the structure of the *Windows* clipboard at the system level. The Clipboard program is a particular class of application known as a *clipboard viewer*, which processes special *Windows* messages and knows how to display data in a number of different formats. The system-level clipboard, on the other hand, is actually a miniature indexed database of pointers to global memory blocks, along with information about the type of data contained in the blocks. The clipboard doesn't have any intrinsic capabilities for processing or displaying data that applications place "on the clipboard," allocating memory to hold such data, or even for validating data by determining whether it is truly in the specified format; applications that use the clipboard rely completely on a set of formatting conventions and on each other's good behavior.

Let's look at a prototypical sequence for putting a chunk of data on the clipboard. The application first allocates a global memory block, locks the block, places the data of interest in the block, and unlocks the block again. It then calls, in succession, the *Windows* functions OpenClipboard(), EmptyClipboard(), SetClipboardData(), and CloseClipboard(). OpenClipboard()'s parameter is the application's window handle, required so that *Windows* can track the owner of the data in the clipboard and keep other applications from changing the data in the clipboard while it is being inspected or altered. SetClipboardData() has two parameters: a "magic number" (reserved identifier) for the clipboard data format and the handle to the global memory block.

EmptyClipboard() and CloseClipboard() do not require any arguments. *Windows*, Versions 2.0 and 3.0, define a number of different "clipboard data formats." These are listed together with their identifiers in Figure 3; applications are also allowed to register and use private clipboard data formats.

When an application wants to fetch data from the clipboard to support a Paste operation, it uses a similar sequence of *Windows* function calls: OpenClipboard(), GetClipboardData(), and CloseClipboard(). OpenClipboard() and CloseClipboard() have already been described; GetClipboardData() accepts the identifier for a clipboard data format and returns the handle to a global memory block or NULL if the specified data format is not available. Once it has the handle, the application can lock the memory block to obtain a far pointer, copy the data to its data segment or to another global memory block of its own, then unlock the block again before closing the clipboard. The application must not free the memory block offered by the clipboard, because this would make multiple Paste operations on the same data impossible; the memory block will be freed automatically by the system at the time of the next call to the EmptyClipboard() function.

A subtle but important feature of *Windows*' clipboard mechanisms is that the Copying application is allowed to make more than one call to SetClipboard-

DDE MESSAGE EXCHANGE BETWEEN EXCEL AND WORD FOR WINDOWS

WinWord → Excel	Initiate	Excel|C:\EXCEL\NICUBEDS.XLC
WinWord ← Excel	Ack	Excel|C:\EXCEL\NICUBEDS.XLC
WinWord → Excel	Advise	AckReq Fmt=Rich Text Format|Chart
WinWord ← Excel	Ack	Negative App=00|Chart
WinWord → Excel	Advise	AckReq Fmt=Text|Chart
WinWord ← Excel	Ack	Negative App=00|Chart
WinWord → Excel	Advise	AckReq Fmt=Printer_Picture|Chart
WinWord ← Excel	Ack	App=00|Chart
WinWord → Excel	Request	Fmt=Printer_Picture|Chart
WinWord ← Excel	Data	Response Release Size=32 Fmt=Printer_Picture|Chart
WinWord → Excel	Terminate	
WinWord ← Excel	Terminate	

Figure 2: DDE message exchange between *Excel* and *Word for Windows*, provoked by a Paste Link operation in *Winword*. This is an edited version of the message log produced by *DDEWatch*, Version 1.3, a shareware *Windows* utility from Horizon Technologies (517-347-0800). The arrows in the left column indicate the direction of the message traffic; the names in the middle column correspond to the *Windows* DDE messages WM_DDE_INITIATE, WM_DDE_ACK, and so on; the right column provides an additional description of the message or the data to which the message refers.

In the first two messages, *Winword* is using the information it got from the CF_LINK entry on the clipboard to initiate a DDE conversation with *Excel*. The next six messages show *Winword* trying to establish a hot link, polling *Excel* with WM_DDE_ADVISE messages specifying various data formats until it finds one that *Excel* is willing to provide. The WM_DDE_REQUEST/WM_DDE_DATA message pair represents the actual transfer of the chart graphic, and the exchange of WM_DDE_TERMINATE messages occurs when *Winword*'s Exit command is selected.

Data()—each with a different clipboard data format—while it has the clipboard open. In fact, a well-behaved *Windows* application is obligated to present, or *render,* the data being copied to the clipboard in as many different data formats as it can, in order to maximize the likelihood that any other application trying to paste the data later will find a format it can use. For example, if you select and Copy an array of cells from an *Excel* spreadsheet and then run the Clipboard viewer, you'll find that not only is the data present on the clipboard as an ASCIIZ string (CF_TEXT, with the cell values separated by tab characters), but it's also present as a bitmap (CF_BITMAP), a metafile picture (CF_METAFILEPICT), and a link (CF_LINK) record that contains the application name, topic name (in this case, the spreadsheet filename), and item (expressed as a range of cell coordinates).

data formats offered on the clipboard in several ways. The IsClipboardFormatAvailable() function can be called with a clipboard data format identifier; it doesn't require the application to have previously opened the clipboard. The function returns a true or false flag only (this is most useful when an application is displaying its Edit menu and must decide whether to "gray out" the Paste command). There is also a CountClipboardFormats() function, which returns the total number of data formats currently held by the clipboard, and an EnumClipboardFormats() function, which allows the application to list, or enumerate, the data formats. Once committed to a Paste operation, the typical application will call EnumClipboardFormats(), inspect the resulting list to find the richest or most complex data format that it can understand, and then import the data.

A well-behaved *Windows* application is obligated to present the data being copied to the clipboard in as many data formats as it can.

At this point— to return to the original different behavior for Paste and Paste L… is obvious. When you request a Paste operation, *Winword* looks in the clipboard for data in a literal format such as CF_TEXT or CF_BITMAP; when you select a Paste Link operation, *Winword* queries the clipboard for a CF_LINK data item that will supply the information necessary to carry out a DDE transaction. The clipboard and DDE, which appeared at first glance to be mutually exclusive, have thus been revealed as being closely interdependent after all. It seems to be an inescapable paradox of graphical user interfaces that mechanisms such as the clipboard and DDE—which appear so natural and intuitive to the user—require the most convoluted and unintuitive coding on the part of the programmer!

DDE PROBLEMS AND LIMITATIONS

Although DDE as an interprocess communications facility is better than nothing, it certainly suffers from some severe problems and limitations. At the user level, DDE is notorious for its fragility and its erratic behavior when the system is heavily loaded. Even the creation of the sim… hot-linked spreadsheet, graph, and doc… ment shown in Figure 1 caused *Winword* to terminate unexpectedly on several occasions, and *Winword* went into a complete panic when I accidentally changed the name of the chart after the hot link had been established.

DDE is also awkward to use because of the "one-way" operation of the DDE linkage. For example, there is absolutely no way for the user to determine, when looking at a Paste Linked graphic in a *Winword* document, what application was responsible for creating the graphic and how to load that application in order to change the graphic.

At the programmer level, DDE is even more of a pain. A DDE transaction demands many exchanges of special *Windows* messages, allocations/deallocations and lockings/unlockings of global memory blocks, and creations/destructions of global atoms. Any of these operations may fail unexpectedly, leaving the DDE transaction in limbo. Or one of the applications in the transaction may suddenly be aborted by the *Windows* supervisor, leaving the other party to the DDE conversation hi… and dry. The complexity of DDE pr… gramming is aggravated by the fact that DDE protocols are largely empirical, based

CLIPBOARD DATA FORMATS AND THEIR IDENTIFIERS

Identifier	Data format
CF_TEXT	Null-terminated (ASCIIZ) text.
CF_OEMTEXT	Null-terminated text in the OEM character set.
CF_METAFILEPICT	Metafile picture structure, which includes the metafile mapping mode, dimensions, and a pointer to the actual metafile data stream.
CF_BITMAP	Device-dependent bitmap.
CF_DIB	Device-independent bitmap.
CF_SYLK	SYLK (Symbolic Link) standard data format (used by *Multiplan, Excel,* and others).
CF_DIF	DIF (Data Interchange Format) standard data format (originally used by *VisiCalc,* now owned by Lotus Corp.).
CF_TIFF	TIFF (tagged-image file format) standard data format for bitmaps (used or understood by many painting, drawing, and screen-capture programs).
CF_PALETTE	A color palette, often used in conjunction with a bitmap specified by CF_DIB.
F_LINK	A series of three ASCIIZ strings identifying a DDE server, topic, and data item, the entire set of strings being terminated by an additional null byte. This clipboard format is registered and used by applications that support warm or hot links.

Figure 3: The identifiers for the standard clipboard data formats used in *Windows,* Versions 2.0 and 3.0. With the exception of CF_LINK, these identifiers are defined in the WINDOWS.H header file in the *Windows* Software Development Kit (CF_LINK is registered on the fly by applications that support hot or warm DDE links). Additional formats are defined for Microsoft's Object Linking and Embedding (OLE), and applications can also register and use private clipboard data formats.

FUNCTION CALLS EXPORTED BY THE DDEML

DdeAbandonTransaction()	Abandons an asynchronous DDE transaction, automatically releasing any resources used in the transaction.
DdeAccessData()	Returns the base address and length of the data associated with a specific data container handle; see also DdeCreateDataHandle() and DdeAddData(). The data is read-only. This call must be followed by a call to DdeUnaccessData().
DdeAddData()	Appends data to the data previously associated with the specified handle.
DdeClientTransaction()	Requests a synchronous or asynchronous data transaction within an existing DDE conversation.
DdeCmpStringHandles()	Given two string handles, performs a case-insensitive comparison of the strings and returns the result (analogous to the C runtime library function `strcmpi`).
DdeConnect()	Creates a DDE conversation with a server, using the specified DDE application name and topic name.
DdeConnectList()	Superset of DdeConnect(); creates a conversation with all available servers that match the specified application name, topic, and language criteria.
DdeCreateDataHandle()	Creates a data container, initializes the container's contents, and returns a handle.
DdeCreateStringHandle()	Creates a container for a string and returns a handle. Unlike data containers, the storage for identical strings is shared between applications.
DdeDisconnect()	Terminates a DDE conversation previously established with DdeConnect() or DdeConnectList(). Any incomplete transactions are abandoned, and any resources they are using are released.
DdeDisconnectList()	Terminates all the DDE conversations previously established with a particular call to DdeConnectList().
DdeEnableCallback()	Enables or disables client or server callbacks. Callbacks are used to notify a client or server of an asynchronous event (for example, the completion of an asynchronous DDE transaction).
DdeFreeDataHandle()	Releases the handle for a data container that was previously allocated by DdeCreateDataHandle().
DdeFreeStringHandle()	Releases the handle for a string container that was previously obtained with DdeCreateStringHandle().
DdeGetData()	Copies the data corresponding to the specified data container handle into application memory.
DdeGetLastError()	Returns more-detailed information about the cause of the most recent DDEML error.
DdeInitialize()	Called by an application to initialize the DDEML library before using any other DDEML function calls.
DdeKeepStringHandle()	Notifies DDEML to maintain a string handle after the client or server exits from the callback routine to which the handle was passed.
DdePostAdvise()	Issued by a server application whenever there is a change to data that has been hot-linked by a client.
DdeQueryConvInfo()	Returns information the server and client engaged in a particular DDE conversation.
DdeQueryNextServer()	Returns the next DDE conversation handle associated with a conversation list created by DdeConnectList().
DdeQueryString()	Returns the string associated with the specified string handle.
DdeNameService()	Used by DDE server applications to register or unregister the application names that they will respond to.
DdeSetUserHandle()	Associates an arbitrary 32-bit value with a DDE conversation handle and transaction. This value's meaning is application-dependent and can be retrieved with DdeQueryConvInfo().
DdeUnaccessData()	Releases a data pointer previously obtained with DdeAccessData().

ure 4: A summary of the function calls exported by Microsoft's new Dynamic Data Exchange Management Library (DDEML). The DDEML hides the details of DDE messages, memory blocks, and global atoms from applications, providing a higher-level, more robust interface for interprocess communication using DDE.

on the behavior of *Excel* and on programmer folklore about what works and what doesn't. Finally, DDE is defined at such a primitive level that a successful DDE transaction demands idiosyncratic knowledge about each application involved. For instance, DDE does not support high-level, abstract, application-independent methods of naming data.

DDE DIRECTIONS FOR THE FUTURE

In forthcoming versions of *Windows,* Microsoft is attacking the traditional weaknesses of DDE at two levels. These innovations will first become evident to most users with the release of *Windows* 3.1, but they are implemented to various degrees in the current versions of *PowerPoint* and *Excel,* and the necessary software tools and libraries are already in the hands of many *Windows* developers.

First, Microsoft has encapsulated the gruesome details of DDE message, global memory block, and global atom handling inside a dynamic link library (DLL) called the Dynamic Data Exchange Management Library (DDEML). This library augments the traditional *Windows* API by exporting a large number of new DDE-specific functions (Figure 4), in effect, redefining the old DDE protocol at a new, higher level. The primitive components of DDE are still present, but they are hidden from the application unless the application wants to know about them; since the underlying technology is unchanged, ''old'' DDE-aware applications can still communicate transparently with ''new'' applications that use DDEML. Presumably, Microsoft has learned some lessons about unstable APIs from its OS/2 experience, and the DDEML functions in 32-bit *Windows* will closely resemble the ones that have been defined for *Windows* 3.1.

Second, Microsoft has defined a new protocol for the construction of ''compound documents'' (that is, documents whose components are created in more than one application) called Object Linking and Embedding (OLE). OLE is built upon DDE, but represents the relationships among document components at a much higher level. As the OLE specification says, OLE is based on a very simple conceptual model: ''There are things and there are places that those things can reside.'' In OLE, the things are called *objects* and the places are called *containers*. When something—formatted text, a graph, or whatever—is copied from one place (container) to another, it isn't plopped

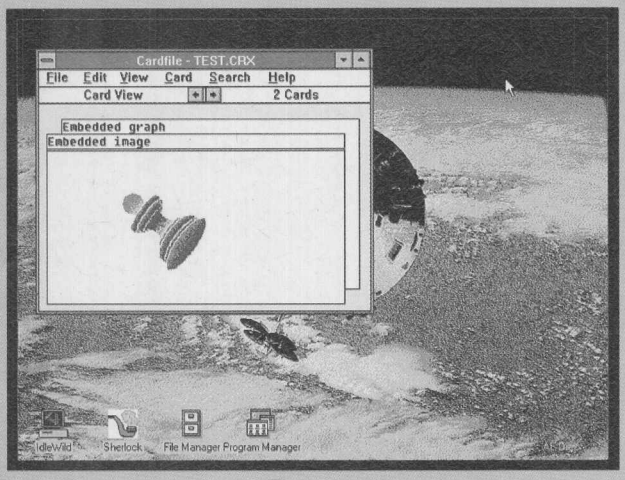

Figure 5: An experimental version of Microsoft's Cardfile application that supports Microsoft's new Object Linking and Embedding protocol (OLE) for compound documents.

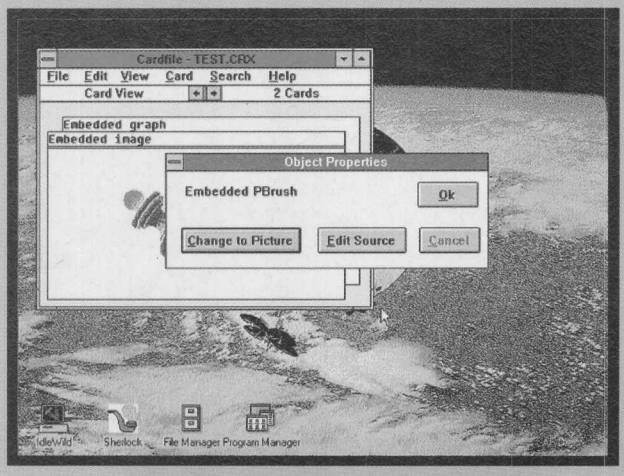

Figure 6: The embedded object that was originally created by Paintbrush was selected by clicking on it. Then the Properties item on Cardfile's Edit menu was chosen, resulting in the dialog box shown here.

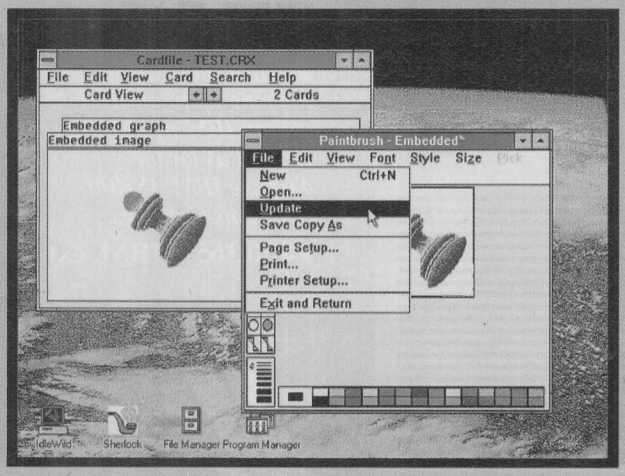

Figure 7: The result after the Edit Source button was clicked. Paintbrush knows that it is functioning as a slave of Cardfile and has replaced its usual File Save and File Save As... menu items with Update (which causes the edited object to be stored back into the Cardfile container).

down in its new location as a bunch of bits that must be understood by the container (or by the application that owns container). Instead, it is embedded in the container as an object that has both content and behavior; the native data is hidden from the container.

What is the behavior of an OLE object? In simplest terms, it's the ability of the object to respond to generic commands (Edit yourself, Show yourself, Resize yourself) that are issued to it by the container. More specifically, the application manipulating the container uses class information associated with the object to load a class-specific dynamic link library that exports a standard set of OLE function calls. The application then calls the library entry point for the desired operation, passing a pointer to the object, and the library translates the generic OLE function request into language that the program that originally created the object can understand. We'll explore OLE in more detail in the next installment of this column; in the meantime, Figures 5 through 7 can give you some inkling of what OLE will mean to users.

These figures show an experimental version of Microsoft's Cardfile application that supports Microsoft's new Object Linking and Embedding protocol (OLE) for compound objects. First, a file containing two cards is opened. One card contains an embedded object created by the OLE-aware graphics tool supplied with *PowerPoint,* and the other contains an embedded object created by an OLE-aware version of Paintbrush.

The object created by Paintbrush is selected simply by clicking on it. Then the Properties item on Cardfile's Edit menu is chosen and a dialog box results. Clicking on the Edit Source button in the dialog box will activate Paintbrush and automatically load the object embedded in the Cardfile container for editing.

Clicking on the Change to Picture button in the dialog box would change the Paintbrush object into a static bitmap and cause it to lose its "behavior," allowing the file to be sent to someone who doesn't have access to Paintbrush.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■